**Michael Lewis**

**Baruch College**

**C++ Programming for Financial Engineering**

**Final Project**

**August 27, 2020**

Project Summary

This application can be used by Financial Engineers to price financial derivatives. In particular, the application is capable of dealing with European and Perpetual American options but can be extended for additional option types.

To use the application, create a European or American option with the required option parameters (T (European only), sig, r, S, K, and b). Afterward, the application provides various functions and methodologies (closed form or finite difference methods) to price the options and to determine their sensitivities (European only).

Importantly, the option data can be sent directly to the console or to a CSV file on the current file path. The latter approach is recommended. The only requirement is to price the option using the function that is pre-built to send data to the CSV file.

Libraries

The application uses STL for Containers and the Boost library to generate the Guassian probability density and cumulative normal distribution functions. As a result, the Boost libraries must be download and included for the application to compile.

Note that there are multiple ways to install Boost:

1. SourceForce: <https://sourceforge.net/projects/boost/files/boost-binaries/>

2. Homebrew: <https://brew.sh/> and run brew install boost

3. MacPorts: [https://www.macports.org](https://www.macports.org/)/ and run sudo port install boost

System Analysis

This application was decomposed into smaller subsystems, with each subsystem having a well-defined responsibility. The class descriptions provided below highlight these subsystems, along with their indications and parameters for implementation.

***AmericanOption***  
The AmericanOption class relies on Template Metroprogamming and Policy-Based Design. Importantly, this is a host class for the various policies (Mesher, Matrix, Output) mentioned below. Please see System Design for additional detail regarding the design choice.

American options can be exercised at any time prior to expiration and generally do not have an exact solution. However, Perpetual American options are the exception because the expiration tends to infinity. Thus, this application appropriately implements the formulae to provide an exact solution for Perpetual American options.

Finally, the pricing functions always return a matrix where the first element of each row is the Call price and the second element of each row is the Put price. This approach ensures that all relevant pricing information is received and makes the system more usable from an analytics perspective.

***EuropeanOption***  
The EuropeanOption class relies on Template Metroprogamming and Policy-Based Design. Importantly, this is a host class for the various policies (Mesher, Matrix, RNG, Output) mentioned below. Please see System Design for additional detail regarding the design choice.

A EuropeanOption is a financial derivative and has the following core member data: T, sig, r, S, K, and b. These options can only be exercised at expiration and, as a result, have prices and sensitivities calculated by the generalized Black-Scholes formulae.

Option sensitivities are the partial derivatives of the Black-Scholes option pricing formula with respect to one of its parameters and, therefore, can rely on closed form solutions for the Greeks in most cases. However, a closed form solution is not guaranteed or can be difficult to find. For those scenarios, the application provides divided difference methods to find a numerical solution.

There is also a relationship between Call and Put prices of a European option. This relationship is defined by the Put-Call parity formula where the Put and Call have the same strike, expiration, and underlying. This relationship can also be tested for a corresponding Put (or Call) price, which helps identify arbitrage opportunities if the relationship is not satisfied.

Finally, the pricing functions always return a matrix where the first element of each row is the Call price and the second element of each row is the Put price. This approach ensures that all relevant pricing information is received and makes the system more usable from an analytics perspective.

***Mesher***  
A Mesher class is a policy used by the financial derivative host classes. It is responsible for creating a one-dimensional domain of mesh points. The mesh points are bounded by [start, stop] and separated by a step size. The Mesher creates an array of a monotonically increasing range for any of the option datum. This mesh array is then fed into the Matrix.

***Matrix***  
A Matrix class is a policy used by the financial derivative host classes. It is responsible for creating a container of option parameters. Each row in the matrix will be identical except for the option parameter that has been monotonically increased by the Mesher.

The container is then consumed by the financial derivative host classes where a Call and Put price is determined for each row of option data. This mechanism allows for the efficient pricing of a wide range of option data that can then be analyzed to show how a change in the single varying parameter impacts Call and Put prices (as well as their Greeks in the case of EuropeanOptions).

***RNG***  
An RNG class is a policy used by the financial derivative host classes. The RNG is responsible for generating the cumulative normal distribution function used to price EuropeanOptions. This class relies on the Boost library.

Additionally, this class provides the normal (Gaussian) probability density function as well as the ability to generate a standard normal distribution using the Mersenne Twister random number generator.

***Output***  
An Output class is a policy used by the financial derivative host classes. The Output is responsible for receiving data matrices from the financial derivative host classes. Upon receiving these matrices, the Output's sole job is to create a CSV file and parse the matrices into rows and columns.

The CSV file is titled either European\_Option\_Data or American\_Option\_Data and is appended with the current date and time to create unique file names across multiple simulations.

Sending data to the CSV file is the recommended approach because it allows for easier analysis, charting, and data sharing. However, the output data can be written directly to the console as well.

See the sample-output folder for an example. This file includes the option Call and Put prices as well as associated option sensitivities (Greeks).

***System Design***

The application implements Template Metroprogamming and Policy-Based Design. These design choices provide several benefits.

For example, host classes derive from each of its policies but do not violate the IS-A relationship because Policy-Based Design inverts the relationship between base class and derived class. An alternative design choice could have been Composition or Aggregation. However, both of these designs would have violated core OOP principles:

1. Composition would have required creating false HAS-A relationships
2. Aggregation would have created objects that survive past the lifetime of a financial derivative even though there is no use for them once the financial derivative has gone out of scope

The policies also encapsulate an orthogonal behavior of the host class, which modularizes the application. Therefore, the application can easily be extended by creating additional policies that subsequently get supplied to the host classes.

As stated, Policy-Based Design relies on Template Metaprogramming. This means the system is parametric polymorphic, which avoids the cost of virtual lookup tables associated with subtype polymorphism.

A screenshot of a social media post

Description automatically generated

**Group A (European Option Pricing)**

After implementing the generalized Black-Scholes formulae, I ran tests for Batches 1-4 and confirmed the output matches the provided solutions. Additionally, I implemented two put-call parity functions. The first is as a mechanism to calculate the call (or put) price for the corresponding put (or call) price and the second as a mechanism to check if the given set of put and call prices satisfy parity. For the latter case, it’s important to set a tolerance to avoid errors stemming from double-precision floating-point formats. In general, a good tolerance is somewhere between 1e-10 and 1e-5 (1e-5 is used in this implementation). The results of these tests are shown below.

A screenshot of a cell phone

Description automatically generated

I also implemented and tested functionality to compute option prices for a monotonically increasing range of a specified option parameter. The specified parameter can be any of the core option parameters (e.g. T, sig, r, S, K, b), which provides flexibility to users of the system. The implementation also correctly handles the relationship between r and b when dealing with stock options and futures. As we know, the generalized Black-Scholes formula requires b=r for stock options, but this relationship does not hold for futures.

To highlight this functionality, I included screen shots below. The first screen shot shows the relationship between call and put prices as we monotonically increase S. The second screen shot shows the same relationship for call and put prices but does so for a generic monotonically increasing parameter (e.g. T, sig, K, and r).

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

**Group A (European Option Greeks)**

To make the system more robust, I also provide functionality to calculate the Greeks. The Greeks are the partial derivatives of the Black-Scholes option pricing formula with respect to one of its parameters. Proof of the formula are beyond the scope of this course and this system provides only a subset of all the possible Greeks. In particular, I implemented formula for Delta (1), Gamma (2), and Vega (3).

1. Delta is the change in the options price due to a change in the underlying’s futures price.
2. Gamma is the rate of change in the options Delta per one-point change in the underlying’s spot price.
3. Vega is the change in the option price per one-point change in implied volatility.

The results of the first test case (question A) is shown below.

A screenshot of a cell phone

Description automatically generated

Because the Greeks are the partial derivatives of the Black-Scholes option pricing formula, we can rely on the closed form solutions (implemented in question A) in most cases. However, a closed form solution isn't guaranteed or can be difficult to find. For those scenarios, I implemented the divided differences method to find a numerical solution.

Similar to the option pricing functionality, these functions are able to handle a monotonically increasing range underlying values so we can study how a change in the asset price impacts its associated Greeks. To highlight the differences between the closed form solution and the numerical method, I included several screen shots below.

Importantly, these tests include a range of values for h, where h is the difference parameter, to highlight how accuracy degrades as h grows larger. In particular, we are accurate to seven to eight places behind the decimal when h is 0.1 but only four to six places behind the decimal when h is 2. This pattern continues as h grows larger and larger. Thus, it’s important to find a value for h that is small enough to ensure high accuracy but not so small that we encounter round-off errors or issues related to the subtraction of quantities that are very close to each other.

A screenshot of a cell phone

Description automatically generated

A picture containing computer

Description automatically generatedA close up of a piece of paper

Description automatically generated

**Group B**

Group B is nearly an exact replica of the functionality I discussed in Group A. The only difference is that Group B implements the formulae to price Perpetual American equity options whereas Group A implemented the formulae to price plain (European) equity options.

As a result, the system can also provide solutions to a monotonically increasing range of option parameters. The specified parameter can be any of the core option parameters (e.g. T, sig, r, S, K, b), which provides flexibility to users of the system.

Notably, a plain option had a strict relationship that required b = r. However, this relationship does not hold for Perpetual American options. Therefore, the implementation also correctly handles this new relationship between b and r.

To highlight this functionality, I included screen shots below. The leftmost screenshot shows the solution to question B, which asked to compare the result of call and put prices as determined by the application versus the solutions provided in the project. The next set of three images shows how the call and put prices change as the specified option parameter is monotonically increased.

A screenshot of a cell phone

Description automatically generated

**Group C**

As expected, increasing the number of intervals or simulations does not guarantee higher rates of convergence. This observation is central to the Monte Carlo method in that the method depends on repeating random sampling. Given this random sampling, it is natural to expect divergence in certain cases despite having larger these larger values for the inputs.

To highlight this observation, let us evaluate the case when there are 100 subintervals and 10M simulations. In this simulation, the Call Price for Batch 1 is slightly worse than it is for the simulation with 100 subintervals and 1M simulations. Similarly, there are also cases when there are higher rates of convergence when the inputs increase in size. For example, the Put Price with 10M simulations versus 1M simulations shows a higher rate of convergence. When taken together, these observations highlight the fact that convergence is not necessarily linear.

With that said, the most accurate simulations tend to have 100+ intervals and 1M+ simulations, which is evident in the images shown below. This observation supports the law of large numbers theorem from probability. The theorem states that the result of a large number of repeated simulations should be close to the expected value and tends to continue this path of convergence as the number of simulations increases.

This finding underlines the importance of developing applications such that we can use these larger numbers in high-performant situations, meaning that we tightly control time complexity and avoid algorithms that might lead to O(N^2).

Batch 4 proved to be an exception, however. When the number of intervals was 10, we saw an increasing rate of divergence when increasing the number of simulations. For all other Batches, increasing the number of simulations had an overwhelming impact on the rate of convergence. To gain a better understanding of why this is happening, I held all parameters constant except for expiry, which was reduced to 1.0. The result of this test suggested that the long time to maturity drove divergence when the number of intervals was small.

We also know that this method requires nested for-loops. It’s important to highlight this observation because there is a common misperception of the time complexity associated with nested for-loops. Often, the observation defaults to “nested for-loops are always O(N^2)”; however, this does not always hold true. Complexity analysis depends on a precise definition of N and there are many cases where having a nested for-loop does not imply O(N^2) (e.g. when the inner for-loop performs a constant time operation). In our case, the number of intervals and simulations are the variables, which we will label N and M, respectively. Thus, the complexity of our MC simulation is O(N\*M). Obviously, when we increase either of these variables, the time complexity increases. For example, the Batch 1 simulation with 1,000 subintervals and 100,000 simulations required ~136 seconds to complete, whereas the simulation with 10 subintervals and 100,000 simulations required only ~1.5 seconds.

For Batch 4, the results are significantly slower to converge relative to Batch 1 and Batch 2, and actually diverge when the number of intervals equals 10. This is likely due to the prolonged time to maturity. It appears that increasing the number of intervals has a greater impact on convergence relative to increasing the number of simulations for Batch 4. For example, the results below show a Call with 10 subintervals and 100K simulations, which has an estimated price of 71.6601 but a price of 89.5241 with 100 subintervals and 100K simulations. When adjusted for only 10 intervals, the result diverges when changing the number of simulations from 1,000 to 10M.

To converge to two places behind the decimal point on a Put requires 1,000 subintervals and 1M simulations. To get the same level of convergence on a Call, we likely need somewhere between 1M and 10M simulations with 1,000 subintervals. As depicted below, those two specific test cases do not achieve the desired convergence, but as we know, convergence is not linear. In almost all cases though, having 1,000 subintervals provides a better estimate than test cases with fewer subintervals, regardless of the number of simulations.

To highlight the above analysis, I included the images below to show the simulation results for Batches 1, 2, and 4 respectively.

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**

**Group D**

A small Standard Error (SE) indicates the sample mean is a more accurate representation of the actual population mean. Additionally, a larger sample size will result in a smaller SE as the statistic approaches the actual value, which we can see in the results shown below. This observation (larger sample size leads to smaller SE) generally correlates with the higher rates of convergence we saw in Group C when the sample sizes increased.

Additionally, we also know that Standard Deviation (SD) is a measure of data dispersion. The results suggest the data has relatively high dispersion rates and is particularly unstable when the number of simulations is small. The prices in Group C showed higher rates of convergence when the number of intervals and simulations grew larger, and we see this same pattern in SD. This pattern tends to occur when the number of intervals is 500+ and the number of simulations is 1M+. Notably, the results for all sizes of intervals tended to stabilize when the number of simulations grew large enough because it is no longer subject to the volatility of a small samples size.

Given this stabilization and the small SE, we can begin to interpret the results as the actual dispersion around the true mean. Thus, the MC method has relatively high rates of dispersion, but we can gain more confidence in the accuracy of our results with larger values because we eliminate the volatility associated with small samples.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**Group E**

Please see the excel file titled “Group E – Excel Visualization” included in the submission folder. I also include a screen shot below as a sample of the output.

A picture containing kitchen, table

Description automatically generated

**Group F**

As we know from the experiments in Group C and D, the Monte Carlo method can have bad time complexity. The FDM approach is significantly more efficient in time complexity. In some situations, the FDM scheme may prove to be a superior solution for the high performant situations, such as those mentioned for Groups C and D.

The FDM method also showed significantly faster convergence than the MC method. To highlight this, I included a comparison of the FDM results with sample results from the MC method in the charts below. In all cases, the FDM scheme showed significantly faster rates of convergence.

However, the FDM scheme was incredibly unstable at small values of N. To see the full range of values, please see the FDM specific files included in the submission folder. In those files, one can visualize how unstable the FDM scheme can be. Owing to this shortcoming, it is important to bound the FDM scheme with properties that prevent input data that cause instability. In this case, we should restrict the size to N >= 10,000 (or N >= 1M for Batch 4 given the longer time to maturity). With the FDM, we no longer have to solve the matrix system, but the step sizes and time intervals must be very small.

Both the MC and FDM schemes converged, in nearly all cases, to two places behind the decimal when compared to the exact solutions. Batch 4 was the only exception, which is in part due to the long time to maturity.

When these various factors are taken in combination and context, it seems clear that the FDM scheme may be the preferred approach as long as additional efforts are taken to ensure stability and accuracy (e.g. small error).

A screenshot of a social media post

Description automatically generated